Using
Automatic Code Generation
In the
Attitude Control Flight Software
Engineering Process

Primary Author: David McComas
NASA Goddard Space Flight Center, Code 582
Greenbelt, MD 20771
301-286-9038
david.mccomas@gsfc.nasa.gov

Co-Authors: James R. O'Donnell, Jr., PhD
Stephen F. Andrews

Abstract

This paper presents an overview of the attitude control subsystem flight software development process, identifies how the process has changed due to automatic code generation, analyzes each software development phase in detail, and concludes with a summary of our lessons learned.

Attitude Control Subsystem (ACS) Flight Software (FSW) and the processes that govern its development are complex. The Microwave Anisotropy Probe (MAP) spacecraft's ACS FSW, currently being developed at the NASA's Goddard Space Flight Center (GSFC), is being partially implemented using Integrated Systems Inc.'s (ISI) MATRIXx which includes an automatic code generation tool AutoCode™. This paper examines the "traditional" ACS FSW development process and describes how the MAP effort, augmented with ISI's tool set, has addressed ACS FSW development complexities.

The MAP ACS team carefully scoped the use of the MATRIXx tools from the outset of the project. The analysts confirmed that MATRIXx was suitable for analysis and algorithm development, but was not certain that AutoCode would be used. Initially, AutoCode's role was to automatically generate an algorithms specification, which has traditionally been a laborious process. If the generated code could be verified, and it passed size and performance requirements, then it would be considered for use as flight code. This low risk approach allowed the team to investigate new technology while addressing the demands of MAP's ambitious schedule with a small development team.

The paper is structured into two sections. The first section provides contextual information for the second section. The first section describes the MAP mission and the flight architecture on which the ACS FSW resides, and the MATRIXx tools. Section two presents an overview of the ACS FSW development process, identifies how the process has changed due to AutoCode, analyzes each software development phase in detail, and concludes with a summary of our lessons learned.

MAP Mission and Flight Architecture Overview

MAP's mission is to probe conditions in the early universe by measuring the properties of the cosmic microwave background radiation over the full sky. These measurements will help determine the values of cosmological parameters associated with the "Big Bang" and determine how and when galactic structures formed. MAP will maintain a halo orbit about the Sun-Earth Lagrange point (L_2), 1.5 million kilometers from the Earth (away from the sun). MAP will maintain a 0.464 rpm spin rate about the spacecraft's Z axis during science observations.

The ACS FSW plays a central role in every phase of the MAP mission using, sensors and actuators to perform attitude determination and control and fault detection and correction. Following launch, the ACS must reduce spacecraft body rates and orient the spacecraft to a power-positive and thermally-safe attitude. The ACS must provide three-axis inertial pointing and provide the capability to slew the spacecraft to new attitudes. The ACS must be capable of doing orbit maneuvers using thrusters. These maneuvers will be used to get to L_2 and to perform L_2 station keeping. The ACS also controls the 0.464 rpm science observation spin.

Figure 1 shows where the ACS FSW fits into the MAP flight architecture. The portion of ACS FSW relevant to this paper resides on the Mongoose processor in the ACS task. The Mongoose uses the software bus for inter-task communication. The software bus is a GSFC custom-built library that provides standardized packet-based inter-task communication and insulates applications from the real-time operating system. A task defines a packet pipe on which to receive data. Task execution is usually controlled by a task pending for data with an optional timeout. The ACS task pends for an Attitude Control Electronics (ACE) sensor data packet, which is generated at a 1Hz rate. After the ACS task receives the packet it converts the sensor data to engineering units in the body frame, updates its attitude knowledge, executes a control law, and issues an actuator command. Each second the ground command packet pipe is polled for new commands, and telemetry packets are generated for onboard storage and/or downlink. Note that the ACS task is isolated from hardware interface details and uses the software bus for all external communications.

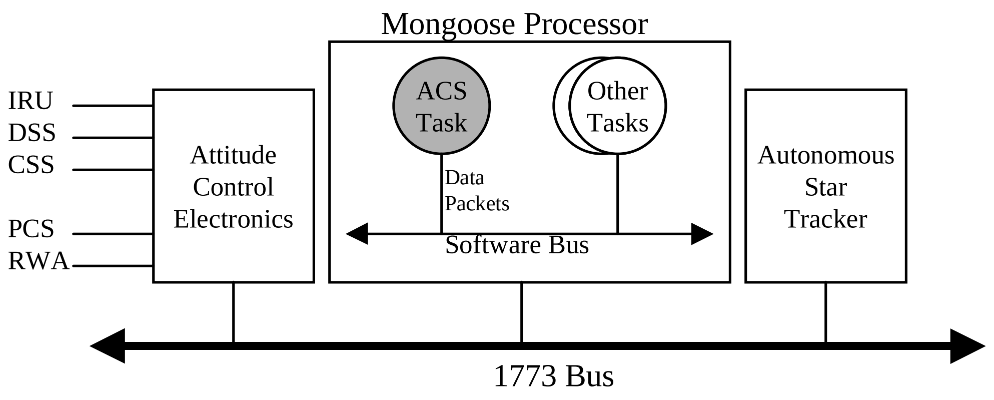

Figure 1 – MAP flight architecture

Two additional ACS features need to be described in order to understand the analysis and design of the automatically generated code and the code that interfaces with it. These two features are the sensors and actuators and the ACS operational modes. MAP uses the following sensors and actuators for attitude determination and control:

Inertial Reference Units (IRU)	Measure changes in MAP's angular position. Spacecraft body rates are derived from the incremental angular measurements.
Digital Sun Sensor (DSS)	Provides accurate measurements ($< 0.01^{o}$) of the sun's position within a 64 degree square field of view.
Coarse Sun Sensors (CSS)	Provide coarse measurements ($< 10^{o}$) of the sun's position. The CSSs are mounted to provide complete sky coverage.
Autonomous Star Tracker (AST)	Provides an estimated attitude derived from star measurements.
Propulsion Control System (PCS)	Provides external force and torque to the spacecraft via hydrazine-fueled thrusters.
Reaction Wheel Assembly (RWA)	Provides spacecraft attitude control via three reaction wheels.

MAP uses five operational modes to achieve its mission goals. Modes are defined in terms of operational objectives, spacecraft control objectives, and performance criteria. Each mode specifies a set of sensors and actuators and a control subsystem configuration.

MAP defines the following modes:

Operational Mode	**Description**
Sun Acquisition (SA)	Uses IRUs, CSSs, and the RWA to acquire a sun-pointing, power and thermally-safe attitude within 20 minutes from any initial attitude.
Inertial (IN)	Uses IRUs, DSS, ST, and the RWA to acquire and hold a fixed commanded attitude.
Observing (OB)	Uses IRUs, DSS, ST, and the RWA to perform a scanning pattern. Observing is the only mode used for collecting science data.
Delta-V (DV)	Uses IRUs and the PCS to perform spacecraft maneuvers. Delta-V is used for trajectory management to get to the Sun-Earth L_2 point approximately 1.5 million km from the Earth (away from the sun) and for L_2 station-keeping.
Delta-H (DH)	Uses IRUs and the PCS to perform momentum unloading.

<u>MATRIXx Overview</u>

The purpose of this paper is not to evaluate MATRIXx. However, it is necessary to understand some of the components and features of MATRIXx in order to understand how they impacted the software development. Figure 2 shows the MATRIXx runtime environment. SystemBuild provides a graphical environment for building models and a graphics package for analyzing data. Xmath provides text-based windows for user interaction and the mathematical computation engine. The three standard user windows are status log, command input, and error message.

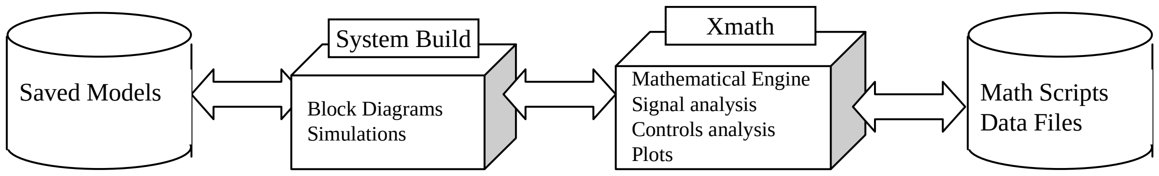

Figure 2 – MATRIXx Environment

An engineer graphically decomposes complex models in SystemBuild using SuperBlocks. SuperBlocks are characterized by their inputs, outputs, and user defined attributes. Timing attributes include continuous, discrete, procedure, and triggered. Procedure blocks can be further classified as standard, inline, macro, interrupt, background, or startup. Attribute details will be described as needed in the paper.

SuperBlocks may contain other SuperBlocks and/or functional blocks. Hierarchies of SuperBlocks are used to abstract system details. Functional blocks cannot be further decomposed. The most common functional blocks used on MAP include trigonometric functions, algebraic functions, logical functions, and dynamic systems functional blocks. Most functional blocks are “closed” which means an engineer can define the block’s I/O and configuration parameters, but cannot change the block’s functionality. For MAP, we used three types of “open” blocks, which allow the user to extend the system’s functionality. Algebraic blocks allow the user to define the block’s outputs as algebraic functions of the block’s inputs (and other parameters). BlockScript blocks allow the user to use BlockScript, a FORTRAN-like procedural language, that allows many programming constructs. Finally, User Code Blocks (UCB) allow user supplied C code to be linked with SystemBuild.

In addition to providing functional organization, SuperBlocks also control some aspects of the data flow. There are three basic methods for transferring data with a SuperBlock. SuperBlock I/O pins can be used to pass data up and down a SuperBlock hierarchy. Read-from and write-to variable blocks can input and output data from a global workspace. The last method, available for most SuperBlocks, is to define SuperBlock parameters. Parameters are variables that are imported from the Xmath variable space and are called percent variables (%VARs). %VARs are used to define variables, such as alignment matrices and controller gains, that do not change during a simulation. AutoCode converts %VARs to global variables that can be written to and read by code external to the automatically generated code. %VARs can be logically grouped into partitions and a SuperBlock can specify a particular partition for its %VARs.

The automatic code generation process is shown in Figure 3. A model must be loaded into SystemBuild and default values for %VARs should be established. This is best achieved by using a MathScript (Xmath command files) to define the defaults. The top most SuperBlock is supplied to AutoCode and code is generated for this SuperBlock and every subordinate SuperBlock. Code can be generated for multi-rate systems using the scheduled-subsystem option, and for single rate systems using the procedures-only option.

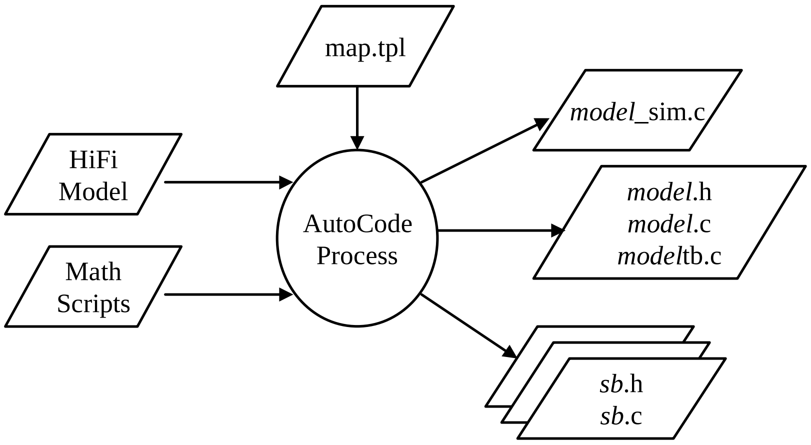

Figure 3 – Automatic code generation process

The code generation process is controlled by a script written in ISI's Template Programming Language (TPL). We extended ISI's default TPL file (renamed to map.tpl) to include the following features:

- Non-FSW is output to a separate *model*_sim.c source file. This includes code such as UCB wrappers (code used to interface C language functions into a simulation via a User Code Block) that could be used in a simulation environment but are not needed in the flight environment.
- Separate header and source files are created for each non-inline SuperBlock. This enhances readability, encapsulation, maintenance, and configuration management. Late changes will result in individual files being delivered.
- Two interface functions *model*_Init() and *model*_Dispatch() are created to provide a consistent interface to the automatically generated code. These functions also provide a placeholder for customizations.

The MATRIXx AutoCode process is essentially closed, which means the code generated for a functional block cannot be altered. This is why the BlockScript and Algebraic blocks are considered to be open since they allow user customization of the generated code. As an example of the closed nature of the code generation process, let us consider how the %VARs are defined. To define the %VARs the TPL script calls a predefined TPL function named define_percentvars(). This generates the code for all of the %VAR definitions. The TPL script can add code around the %VAR definitions and it can control what file the definitions appear in, but that is all it can do.

ACS FSW Development Process

Figure 4 illustrates the MAP ACS FSW development process. The solid lines represent the traditional process and the dashed lines indicate where the process has significantly changed for MAP. Many traditional parts of the process were impacted as well, and these impacts will be described in their respective sections. Note there is an iterative aspect to our development process that is not shown in the figure. The FSW is delivered to the build test team in incremental builds and the requirements and design are continually refined throughout the process.

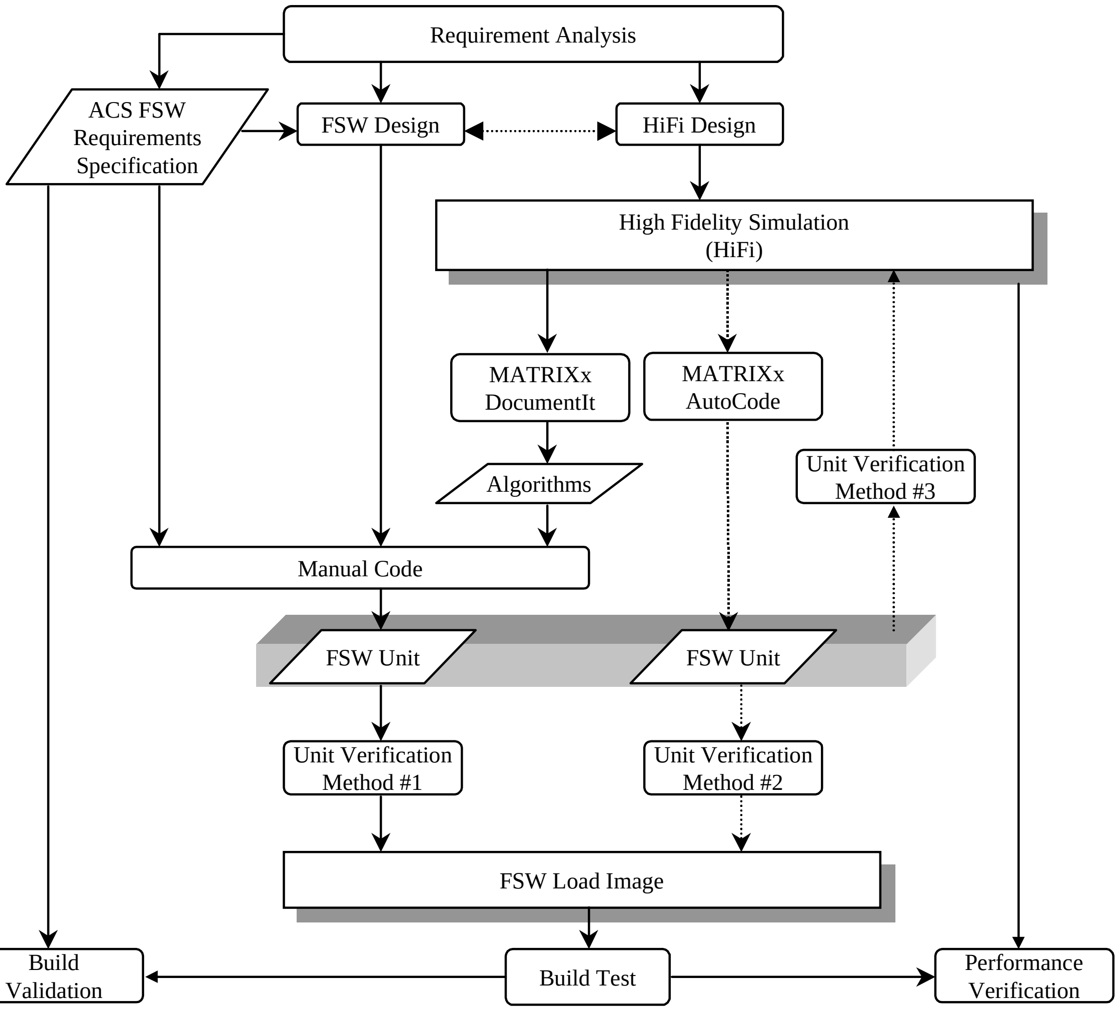

Figure 4 - ACS FSW Development Process

The process begins with ACS FSW requirements analysis. This is a system engineering process primarily involving Guidance Navigation and Control (GNCC) analysts, FSW specialists, and spacecraft engineers. This activity produces an ACS FSW Requirements Specification and feeds directly into both the FSW and high-fidelity (HiFi) simulator designs. HiFi is required for the GNCC analysts to validate the spacecraft controls algorithms which are needed in the ACS FSW. On previous GSFC missions, the ACS FSW architecture and the HiFi software architectures were developed independently, with minimal-to-none commonality between the two software systems.

The use of AutoCode requires that the FSW and HiFi architectures account for one another's environment. The graphical HiFi environment encapsulates function and data into a component called a SuperBlock. In order for a translated SuperBlock to exist in the FSW environment, HiFi must emulate the FSW environment or the SuperBlock must avoid interfacing or relying on features of the FSW environment. Any differences that exist between the two environments must be accounted for by the automatic code generation process or by manually changing code after it has been generated. The analysis and design section describes the translation strategies that were taken on MAP.

The implementation phase gets its inputs from the ACS requirement specification and the ACS algorithm specification. The ACS requirements specification defines what the FSW needs to accomplish in terms of functional and performance statements. The algorithms specification is a companion to the requirements and it defines the mathematical details needed to be implemented by the FSW in order to meet the functional and performance criteria. Traditionally all of the FSW has been manually coded from these inputs. Two significant changes were made on MAP. First, the generation of the ACS algorithms specification was automated using ISI's DocumentIt^{TM}. In the past, the generation and maintenance of the ACS algorithm specification was a tedious job that has required a dedicated analyst. Second, the automatic generation of some of the flight code eliminated part of the manual coding effort. These two changes were not free and the price of using the tools will be discussed in the implementation section.

Testing occurs at both the unit and build test levels. At the unit level, 3 verification processes have been enumerated. Unit verification method #1 requires the developer to verify that the requirements and algorithms have been implemented correctly. Depending upon the scope of the algorithms being coded, unit test data from HiFi may or may not be supplied. Unit verification method #2 is new and is used to verify the automatic code. Since the HiFi and FSW have a common interface to the automatic code it was relatively easy to capture HiFi data at the common interface and feed it through the automatic code. Even without AutoCode, this method could have been employed in the past if the FSW and HiFi designs used a common architecture. Unit verification method #3 was an unanticipated benefit of using the tool set. We were able to take the entire FSW attitude determination subsystem flight code and run it in the HiFi.

At the build test level, the changes were mostly due to other features of the tool set beyond AutoCode. The main advantage of the tool set is that the analysts used the HiFi data analysis platform for test data comparison between data from the development lab and data from the HiFi. In the past, the build test data analysis platform was not necessarily the same as the HiFi's platform. Having the data analysis platform integrated with the HiFi platform also enabled the script files developed during analysis to be used for build test verification.

Analysis and Design

The analysts and programmers had to coordinate their efforts during the analysis phase because automatic code generation requires a tight coupling between the HiFi design and the ACS FSW design. First, both groups needed to understand the capabilities of the tools in order to understand how to best utilize them. Concurrently, a preliminary architecture that would be common to both the HiFi and the FSW needed to be defined. The definition of this architecture was based on MAP requirements analysis and on previous mission architectures. The common architecture identified major components such as sensors, actuators, control modes, attitude determination, and ephemeris and the data that flows between the components. With detailed knowledge of the tools and a preliminary architecture, the MAP team was prepared to define the scope of the automatically generated code. Business forces as well as technical forces shaped the boundary of the automatically generated code.

The primary business issues are that we meet the schedule while delivering a high quality, testable product that implements the requirements. MAP is Goddard's first mission to use an automatic code generator for its FSW, and prior to MAP, Goddard had no experience with ISI's code generator. Our strategy towards mitigating risks was to minimize the impact of the FSW environment upon the HiFi and to limit the scope of the automatically generated code to a portion of the ACS FSW that has a high algorithm-to-code ratio. This strategy minimizes the impact to the analysts while taking advantage of the biggest benefit of AutoCode, which is to eliminate the error prone process of manually translating algorithms to flight code.

Technically, automatic code generation is the translation of a design from the HiFi environment to the FSW environment. The HiFi environment must model any aspects of the FSW environment if the generated code is to be linked with the flight code without any manual changes to the automatic code. As described in the MAP flight architecture overview, MAP is using the software bus as the inter-task communication medium. Opting not to model the software bus in HiFi, we immediately limited the scope of each AutoCode invocation to an intra-task scope and ISI's real-time operating system, pSOSystem, was not even considered. There are also unique flight software interfaces for processing ground commands, generating asynchronous event messages, and notifying the Fault Detection and Correction (FDC) subsystem. Again, we opted not to model these interfaces in the HiFi, further restricting the AutoCode scope.

With these guidelines in hand, we were prepared to identify the portion of the ACS FSW that is automatically generated. Figure 5 shows a simplified high-level block diagram of MAP's flight control software. This software is suitable for a single task because all of its components execute at 1Hz and have fairly strong data cohesion. Sensors measure spacecraft position and rates. Attitude determination uses sensor measurements to update the onboard estimated attitude, which is supplied to the controller subsystem. The desired spacecraft attitude is either supplied by mode management or internally computed by command generation. Attitude error computes control errors for the control law based on a combination of sensor measurements, estimated attitude, and commanded attitude. The control law computes control torques which are output to the actuators.

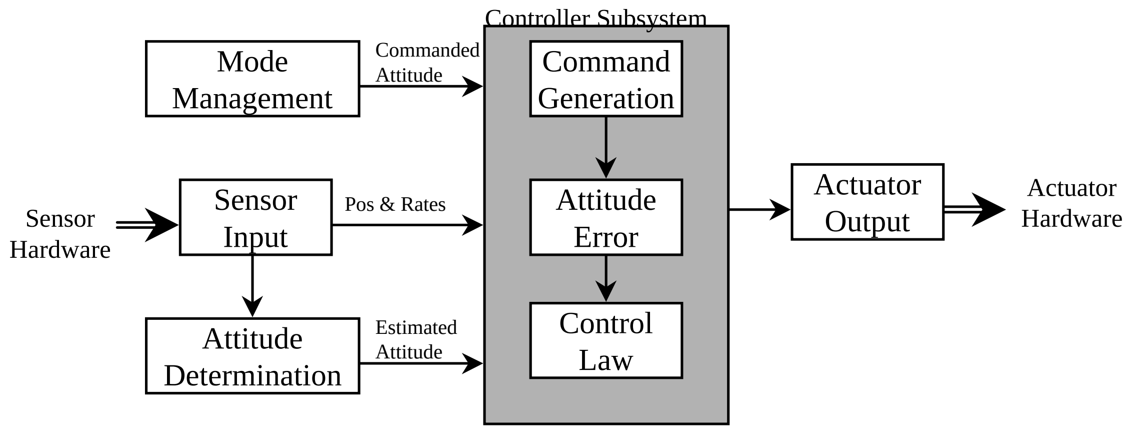

Figure 5 – ACS FSW block diagram

The shaded controller subsystem identifies the portion of the MAP ACS FSW that is being automatically generated. This final design takes advantage of the controller subsystem's relatively small and simple set of inputs and outputs. The controller subsystem's local rotating-sun-reference coordinate frame is encapsulated entirely within the AutoCoded portion of the FSW. Since the controller subsystem components execute at the same rate, we can use the "procedures-only" AutoCode option, which greatly simplifies the automatic code. Attitude determination shares many of the same attributes as the controller subsystem with respect to being suitable for AutoCode, but it was not chosen to be automatically generated since MAP could adapt an existing attitude determination subsystem from a previous mission.

Implementation

Once the scope of the automatically generated code was defined, we turned our attention to the code that is generated and to the manual code that interfaces with the automatic code. Figure 6 is a class diagram showing the classes involved with the manual-to-automatic code interface. AutoCode creates a collection of C functions that can be conceptualized as a single object. This object, named achifi, corresponds to the parent SuperBlock supplied as an input to AutoCode. Achifi provides two interface functions achifi_Initialize() and achifi_Execute(). Outputs from achifi are exported via a global data structure named achifi_Output. The manual code treats this as a read-only data structure; although no mechanisms enforce this rule.

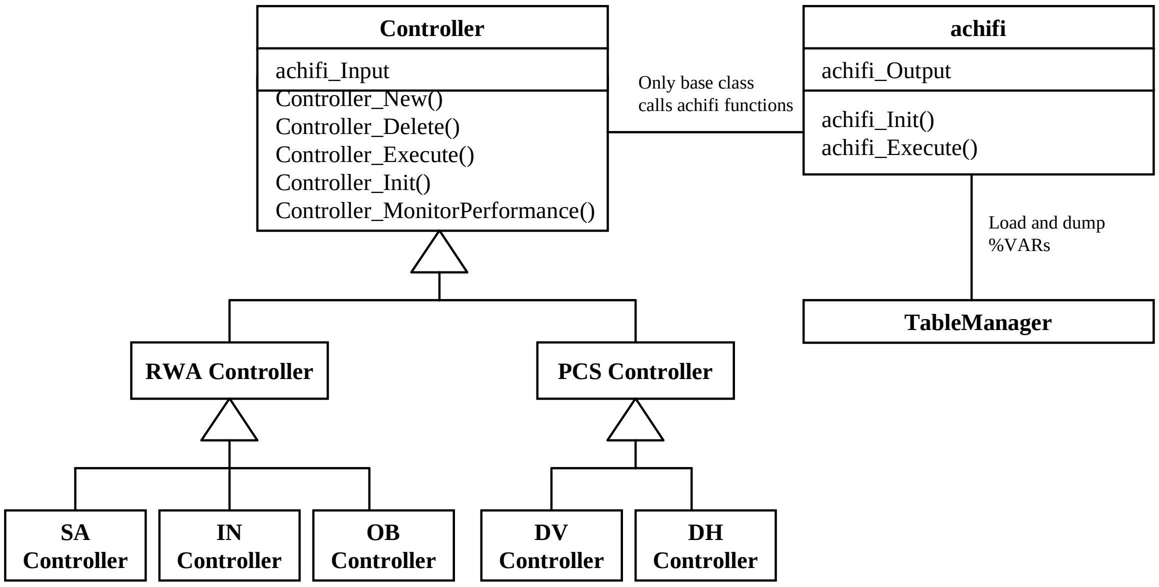

Figure 6 – Manual and automatic code class diagram

The manual code is designed as a class hierarchy designed according to the operational modes identified in the mission overview section. The base controller class defines functions and data that are common to all controllers. Next, the control modes are classified according to which actuator is used for control. Finally, individual modes are defined. The base controller class is the only class that invokes achifi's member functions. Achifi_Initialize() is called when the processor is initialized and whenever a mode transition is performed. Achifi_Execute() is called during each control cycle. Achifi_Execute() manages calling the other functions created by AutoCode.

The object-oriented design was implemented in C by constructing virtual function pointer tables and the design has proven to be very robust with regard to automatic code interface modifications. Most automatic code dependencies are encapsulated by the base controller class, so changes to the interface have had no ripple affect. The base class also provides functionality that can be shared by all controllers. Controller_MonitorPerformance() monitors the performance of achifi by monitoring body rates, attitude errors, and body rate errors. Controller unique performance limits are supplied to Controller_New() when a controller is instantiated. Another benefit of the object-oriented design is that the class hierarchy has resulted in small, easy to test functions. Below are some excerpts from the ACS FSW showing how the base controller class manages the automatic code.

Controller file scope excerpt
```
#include “achifi.h”                /* Autocode header file  */
achifi_Input_Rec  achifi_Input;    /* Autocode input record */
```

```
Controller_Init() excerpt
achifi_Init(&achifi_Input,TRUE);   /* TRUE means copy %VARs from EEPROM */


Controller_Execute() excerpt
achifi_Input.BodyEstRateX = DataMgr.Config.BodyRate.Comp[X];
achifi_Input.BodyEstRateY = DataMgr.Config.BodyRate.Comp[Y];
achifi_Input.BodyEstRateZ = DataMgr.Config.BodyRate.Comp[Z];

achifi_Input.Rwa1MeasTachSpeed = RWA_ProcData.Speeds[0];
achifi_Input.Rwa2MeasTachSpeed = RWA_ProcData.Speeds[1];
achifi_Input.Rwa3MeasTachSpeed = RWA_ProcData.Speeds[2];
. . .
achifi_Execute(&achifi_Input,DataMgr.Config.DeltaTime);

AttCtl.BodySysMom.Comp[X] = (float)achifi_Output.BodyMeasSystemMomX;
AttCtl.BodySysMom.Comp[Y] = (float)achifi_Output.BodyMeasSystemMomY;
AttCtl.BodySysMom.Comp[Z] = (float)achifi_Output.BodyMeasSystemMomZ;
AttCtl.BodySysMomMag = (float)Vector3f_Mag(&AttCtl.BodySysMom);
. . .
```

The automatic code is less readable than the manual code shown above, but this is mostly due to the inclusion of SystemBuild's numeric block identifiers in the variable names. The automatic code generation is very systematic and once you get a feel for how status information, state information, inputs, outputs, and initialization are managed, the code is is relatively easy to read. In fact, comments are inserted in the code to identify which block is being coded. The code's readability is further enhanced by having each SuperBlock output to a separate file, using data naming conventions in the HiFi, and labeling all block I/O lines in the HiFi.

There are a few drawbacks to the automatic code, but none of them have proven to be fatal. The automatic code is large and less efficient than its manual equivalent. We have not had the luxury of duplicating the coding effort manually, but there have been a couple of cases when a fair comparison between the manual and automatic code could be made. In these cases the automatic code has been two to three times larger than the manual code. This code bloat is primarily due to the fact that AutoCode does a lot of data copying before and after calling a procedure. Declaring procedure blocks inline can reduce this overhead, but this doesn't allow functions to reside in individual files. Since MAP has a 1Hz ACS task execution rate and 4 megabytes of EEPROM, resources have not been a concern.

There are a few non-resource issues that did create some trouble. We wanted to treat %VARs as one or more FSW tables. A FSW table must contain physically contiguous data and we typically achieve this by defining a table as a C structure. Unfortunately the Xmath variable partitions do not translate into C structures. We were able to contiguously group the %VARs by defining the %VARs in a separate file. Using linker scripts, the %VAR object file was defined as a contiguous data structure.

There have been two cases when the generated code wasn't what we expected. In both cases, default values were hard coded for %VARs instead of variables being used. ISI's response was that these are features of the blocks in question, although that documentation has not been found yet. Hard coded %VARs is unacceptable so we had to develop workarounds. In one case we coded the block in BlockScript and in the other case we changed the design so the offending block wasn't used. Aside from the %VAR problem, the generated code has properly implemented the HiFi design.

The last issue with the generated code concerns the time between valid control cycles, which we refer to as delta time. Nominally, delta time is one second, but there are situations in which this time can be greater than one second. The ACS FSW must use the actual delta time to safely control the spacecraft. However, AutoCode hard codes a one second delta time because the top-most SuperBlock is defined as a 1Hz procedure block. To correct the situation the automatically generated code must be manually changed to use the delta time passed to achifi_Execute(). Since our automatic code generation process produces a separate source file for each SuperBlock, a manual code change has to be made only once, unless the SuperBlock changes in a future build.

Testing

Both unit testing and build testing have been improved by using MATRIXx. Many of these gains are the result of the entire tool set, not just AutoCode, and are also the result of the FSW developers and analysts working more closely together than on previous Goddard missions.

Figure 7 shows the primary unit testing effort. The HiFi outputs simulation results, simulation inputs, and the automatic code to the unit test platform. The unit test (UT) driver is linked with the FSW controller classes and the automatic code. Five HiFi test cases, one for each operational mode, were used as the test suite. This data was run through the FSW and the FSW results were compared to the HiFi results. This testing has consistently shown that the automatic code accurately represents the Systembuild design.

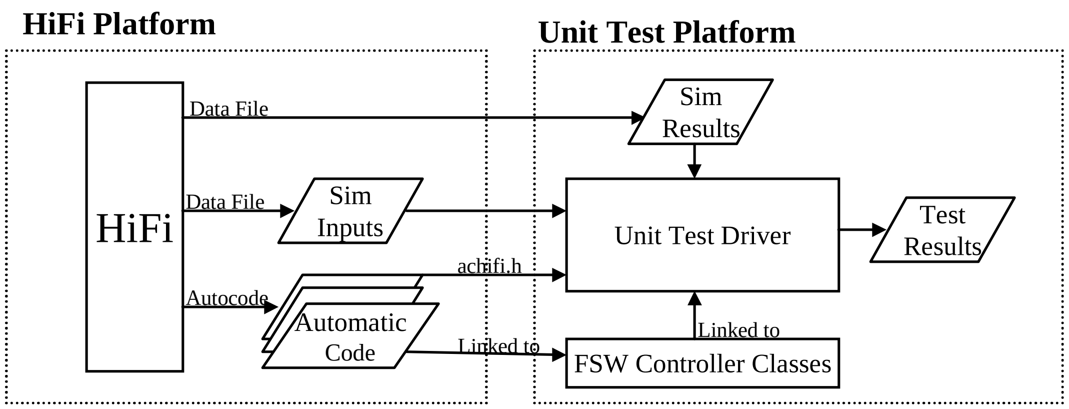

Figure 7 – Automatic code unit test

Figure 7 identifies the primary unit testing activity, but there was additional unit testing performed. A separate UT was developed to perform boundary testing, full path coverage, and to test miscellaneous items such as whether %VARs are properly parameterized. This additional testing is important because the analysts have not traditionally programmed for flight and have not had to worry about issues such as protecting against dividing by zero. This is also our first experience with AutoCode and we don't have past experiences to provide a basis for confidence in the tool. This is the testing that found that the %VARs are not being parameterized for some of the blocks.

Several additional features of the UT process should be noted:

- The UT calls the controller object functions in the same manner as the FSW component that manages the controller objects. This testing verifies that the manual code and the automatic code are properly integrated.
- Achifi.h is shown as a separate input into the UT because it is parsed by the UT to identify the variable offsets within the simulation data file. This aspect is useful because this allows the order of the data within the simulation data file to change without impacting the UT. Since the simulation data file contains data used by the analysts to verify the HiFi, this feature allows the analysts to change what data gets captured without worrying about preserving the order of the data for the UT.
- Since MathScripts managed and documented HiFi test cases, the FSW developers didn't require as much of the analyst's time as in the past to get necessary information. Defining common controller interfaces in both the HiFi and the FSW and having a standardized data file format were the drivers that empowered the UT. These activities could occur with or without automatic code generation.

The improvements made to build testing were achieved by extending the tool set. Build testing involves verifying the FSW's performance in the breadboard lab. Traditionally, a small set of performance test cases were identified. These test cases were executed in both the HiFi and in the build test lab. Two common problems with performance verification involves setting up the same test case in both environments and comparing the results. Through a combination of UNIX scripts and MathScripts, an automated process was developed to transform build test scripts and data sets into HiFi MathScripts. The MathScripts are executed to generate HiFi performance data for the corresponding build tests. The ability to be able to generate these tools was also facilitated by the fact that the HiFi design is similar to the FSW design so the HiFi can be configured in a manner similar to how the FSW is configured via table loads and commands. Another set of MathScript tools were created to automate the generation of comparison plots. The plot generation and comparison tools use GUI-based menu systems making them easy to use. These tools provide a consistent and efficient means for all team members to generate and analyze data without having to dedicate an engineer to these tasks.

Lessons Learned

The most dramatic change has occurred to our ACS FSW development process. Figure 8 shows the same software development process shown in Figure 4 but in terms of activities and products. What has changed is that many people are performing multiple roles. The analysts have participated in every activity. They perform time domain analysis using the HiFi, help to write the requirements, attend code reviews held by the developers, write build test procedures, and serve as the focal point for performance verification. Likewise, the developers have help to write the requirements, reviewed the portion of the HiFi that is being translated to FSW, developed code, and supported performance verification. We have intentionally not allowed the developers to participate in build testing for independent verification. The tools have brought the team closer together but also allowed them to work more independently and efficiently. Many process participants have worked as system engineers with a specialty.

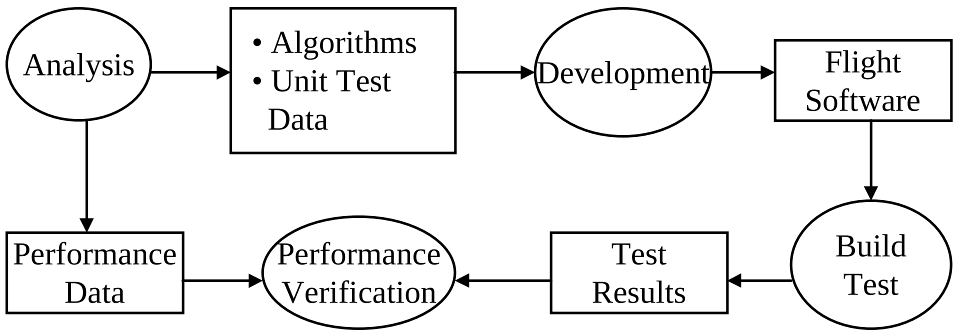

Figure 8 – Software development roles

Trying to quantify these perceived benefits is not so clear cut. We only have limited metrics for the Rossi X-Ray Timing Experiment (RXTE), spacecraft which is a recent Goddard ACS FSW development effort. The following data is a simple comparison of the RXTE and MAP ACS FSW development efforts.

Lines Of Code(LOC)	Spacecraft	Man Years (MY)	LOC/MY
33,318	XTE	13.8	2414
17,525	MAP	6.1	2872

This data appears to show that MAP has been slightly more productive. However, there are many factors to consider when evaluating this data.

- We only have metrics on how much time a developer spent on a project. These metrics are not refined enough to know how much time a developer spent on activities other than development.

- Conversely, the man year data doesn’t include analysts or build testers, yet one of the empirical benefits of AutoCode is that it improves the entire development effort. We definitely reduced the manpower required to translate the HiFi to FSW by using AutoCode, and on RXTE, an analyst was dedicated to creating, executing, and plotting HiFi performance runs for build test verification. These benefits have not been quantified by the data above.
- As stated before, the automatic code was two to three times larger than if it were manually coded. The 5,356 lines of automatic code (31% of the MAP ACS FSW) inflate the MAP production rate.

This data is very encouraging considering that MAP incurred a learning curve with the new tool set. Unfortunately we don’t have any build test metrics, but the test effort has been observably better than on RXTE and we do know the lab is vacant on weekends!

In summary, our low risk approach was successful in allowing us to investigate new technology while meeting the demands of MAP’s ambitious schedule with a small development team.